Protocol Audit Report

Version 1.0

Securigor.io

December 31, 2023

Protocol Audit Report

Securigor.io

December 31, 2023

Prepared by: Securigor Lead Auditors:

- bigBagBoogy

Table of Contents

Protocol Summary

Protocol: A smart contract applicatoin for storing a password. Users should be able to store a password and then retrieve it later. Others should not be able to access the password.

Disclaimer

The Securigor team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond to the following commit hash:

```
7d55682ddc4301a7b13ae9413095feffd9924566
```

Scope

```
./src/
#__ PasswordStore.sol
```

Roles

- Owner: The user who can set, and read the password.
- Outsiders: No one else should be able to set or read the password.

HIGH

[1-H] Storing the password on-chain makes it visible to anyone

Description: All data stored on-chain is visible to anyone, and can be read directly from the blockchain. The `PasswordStore::s_password` variable is intended to be private variable and only accessed throught the `PasswordStore::getPassword` function, which is intended to be only called by the owner of the contract.

We show one such method of reading any data off chain below.

Impact: Anyone can read the password, severely breaking the functionality of the protocol.

Proof of Concept: The below test case shows how anyone can read the password directly from the blockchain

1. Create a locally running chain

```
make anvil
```

2. Deploy the contract to the chain `make deploy`

3. Run the storage tool

We use `1` because that's the storage slot of `s_password` in the contract

```
cast storage <address here> 1 --rpc-url http://127.0.0.1:8545
```

You'll get an output that looks like this: `0x6d7950617373776f726400000000000000000000000000000000000000000`

You can parse that hex to a string with:

`cast parse-bytes32-string 0x6d7950617373776f72640000000000000000000000000000000000000000`

and get an output:

`myPassword`

Recommended Mitigation: Due to this, the overall architecture of the contract should be rethought. One could encrypt the password off-chain, and then store the encrypted password on-chain. This would require the user to remember another password off-chain to decrypt the password. However, you'd also likely want to remove the view function as you wouldn't want the user to accidentally send a transaction with the password that decrypts your password. –> Also MEV frontrunning!

[2-H] `PasswordStore::setPassword` has no access control, meaning a non-owner can change the password

Description: the `PasswordStore::setPassword` function is set to be an `external` function however, the natspec of the function an overall purpose of the smart contract is that `This function allows you to store a private password`

```
 function setPassword(string memory newPassword) external {
@>    // @audit - There are no access controls
        s_password = newPassword;
        emit SetNetPassword();
    }
```

Impact: Anyone can set/change the password of the contract. Severely beaking the intended functionality of the contract.

Proof of Concept: add the following to the `test/PasswordStore.t.sol` file.

```
function testAnyoneCanSetPassword(address randomAddress) public {
        vm.assume(randomAddress != owner);
        vm.prank(randomAddress);
        string memory expectedPassword = "myNewPassword";
        passwordStore.setPassword(expectedPassword);

        vm.prank(owner);
        string memory actualPassword = passwordStore.getPassword();
        assertEq(actualPassword, expectedPassword);
    }
```

Recommended Mitigation: Add an access control modifier to the setPassword function.

```
if (msg.sender != s_owner) {
    revert PasswordStore__NotOwner();
}
```

LOW

[1-L] The `PasswordStore::getPassword` natspec indicates a parameter that doesn't exist.

Description:

```
    /*
     * @notice This allows only the owner to retrieve the password.
@>   * @param newPassword The new password to set.
     */
    function getPassword() external view returns (string memory) {
```

The natspec for the function `PasswordStore::getPassword` indicates it should have a parameter with the signature `getPassword(string)`. However, the actual function signature is `getPassword()`.

Impact: The natspec is incorrect.

Recommended Mitigation: Remove the incorrect natspec line.

```
-     * @param newPassword The new password to set.
```